

Object Oriented Programming and C++

Object Oriented Programming and Abstract Data Types

Object Oriented Programming (OOP) is a programming paradigm based on Abstract Data Types (ADT) . ADTs are models for a certain class of data structures encapsulating both data fields and methods for accessing them. Roughly speaking, with an ADT you can represent any “real” object in the means of how it is internally made vs. the way outer world interacts with it. (SW Engineering professors often think to programmers as “philosophers”)

“In computing, an abstract data type is a model for a certain class of data structures/types that have similar behavior/semantics. An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.”(wikipedia.org)

The key example is the Counter. What do you expect from a counter?

- Initialization
 - `mycounter = 0`
- Increment
 - `mycounter++`
- Read its value for some functionality
 - `print(mycounter)`

If you think about it, those 3 operations are everything you need from a counter from the “customer” point of view. In OOP your focus must be on how you INTERACT with it.

Of course you can implement a counter in any programming language/paradigm. So why would you want to do it with OOP?

The OO-approach splits a big problem into small sub-problems and models them (and the relationships between them) using ADTs. This greatly tackles the complexity of the problem and lets you to program by the means of “black boxes” which can be developed and tested separately. Intuitively, think to the counter: once it is developed and tested, you can re-use it with no modifications. But there are plenty (and more) reasons ...

C++ Classes and Objects

C++ (Bell Labs, 1979) is an object oriented, statically typed language. It is an evolution of C (up to 1983 it was called “C with Classes”), thus the syntax is C-like. C code (can) also run on a C++ compiler. Programming OOP with C++ you define ADTs called **Classes** and then you instantiate them creating **Objects**.

```
/* Declaration of the Counter Class */
public class Counter { /* Class body */ }

/ * Declaration and instantiation of a Counter Object */
Counter *c = new Counter();
```

Writing/implementing a class means

- Designing its internal representaton (in the Counter example, an integer to store the current value/state of the counter)
 - `int val;`
 - each object method accesses its object fields via the `this` keyword
- Providing a way to instantiate (and destroy) the Counter Objects: Constructors and Destructors
 - Same name of the class; NO return type; optionally, args (can do *overloading*)
 - `Counter(){ this->val = 0; }`
 - `~Counter() { /* In this example, we do nothing */ }`
- Providing a way to interact with it: its interface towards the external world!
 - `void increment() { this->val++; }`
 - `int read() { return this->val; }`
- Golden rule: Class names start with capital letter
- “Good” programming rule: a class named `Counter` SHOULD be put in a file called `Counter.cpp/h`

EXAMPLE

Note that the internal members of a class are (should be) declared with the **private** specifier, that means you cannot access them externally (this is called **encapsulation** and is very important, since it gives you the full control on the state/internal variables of the system). On the contrary, the **public** specifier identifies for instance public methods (use it only for them!), i.e. accessible from everyone.

- A “good way” of programming is to provide two methods (the so-called *getter /setter*) to get/set the value of the `private` fields (of course only if needed). Those methods should be called
 - `int getVal(){ return this->val; }`
 - `void setVat(int newVal) { this->val = newVal; }`
 - You keep control on the state of the object (value of the fields)

EXAMPLE

Inheritance and Polymorphism

In C++, code reuse is achieved by *separating* the data definition (in a header file: `Counter.h`) by the implementation (`Counter.cpp`). By compiling code and creating a static/dynamic library, you will provide your component/class *without -for instance- publishing the code* (=> business, no need to recompile, or even linking in case of dynamic libs).

Code reuse does not only mean “write-once-use many-times”, but has more interesting implications. For instance, imagine that you want to create another `CounterDec` with the capability of decrementing its internal value. This ADT is an “improved” counter, but it’s still a `Counter`. Instead of rewriting (or cutting/pasting) the old code, you may want to **extend** the old code with the new functionalities. Consequently, you create a new class **inheriting** all the functionalities by the old `Counter` class.

```
/* CounterDec extends Counter */  
class CounterDec : public Counter
```

The new `CounterDec` is still a counter, and exposes all the functionalities of the `Counter` class.

- You can add new functionalities to the `CounterDec`
 - `void decrement() { this->val--; }`
- You can even redefine base class functionalities!! (**Method overriding**)
 - `void increment() { this->val += 2; };`
- The problem: the `val` field is specified as `private`. This means that NO-ONE can access it (..either a subclass!!)
 - Solution: `protected` specifier: “private...but not for subclasses”

EXAMPLE

Method overriding lets us to introduce **Polymorphism**. *Polymorphism is the capability of one type to be used as another type.* For instance, by overriding the `increment()` method, we can use `CounterDec` as a `Counter`, but its behaviour will be the redefined (overridden) one. It is a very powerful mechanism and greatly simplifies programmer’s life (ok...after the programmer “gets it”)

EXAMPLE

Interfaces

Interfaces allows you to specify the way an object interacts with other objects. It is a definition of the methods (NO implementation) for this (family of) objects. The interface specifies the so-called *contract*, or *protocol* to interact with such an entity.

In C++ an interface is a class whose members are declared as `virtual`. The `virtual` keyword forces the binding to the method to be dynamic (delayed). Dynamic binding → polymorphism!! Moreover, (and of course) to have polymorphism working you MUST access to the object only via pointers and not in a statical fashion.

In our example, the interface `ICounter` exposes TWO methods (constructors, since they are implicit and mandatory in any object, cannot be declared `virtual`)

- `virtual void increment() = 0;`
- `virtual int read() = 0;`

EXAMPLE

C/C++ Macros

The simplest definition of “macro” is “a portion of code identified by a name”. Whenever this name is found, and in PRE-COMPILATION phase, the macro is *expanded*, that is, the name is simply (and stupidly) replaced with the macro body. A very simple example

```
/* Defining a portion of code (5) with the name N */
#define N 5

void main()
{
    int a = N;
    printf("%d\n", a); // Prints the value '5'
    printf("%d\n", N); // This also prints the value '5'
    printf("N\n");    // Warning! This prints the 'N' letter ...
}
```

A macro can be used to identify any portion of code...this code will be copied as-is to replace the macro name, before the code is compiled.

EXAMPLE

Macros accept parameters. Anyhow they are NOT functions. In a macro code, parameters are just placeholders for the values you insert.

- They are a very powerful mechanism to organize code and speedup performance.
 - Smaller code, no code repetitions
 - A function call is more onerous (due to context switch) than a macro since the latter is just code (from the program point of view)
- ...but remember. A macro is just *a portion of code with a name*: you can do the nastiest things with macros!!
 - Even “assembling” the name of variables/methods and so on....
 - You CANNOT debug a macro the standard way you used to!!
 - => Use them for small and well tested portions of code (i.e. constant values as π , simple functionalities as SIN(a), MAX(a,b), etc...)
- The good/bad news: SystemC is a MACRO LIBRARY :)

EXAMPLE

Read from *stdin*, Write to *stdout*

`stdio.h` lib (automatically included by compiler) provides you with the capability of interacting with Input (Keyboard+Shell) and Output (typically the Shell).

- To **read** from *stdin*:
 - `scanf("%d", &a);`
 - `%d` => means you are reading an integer

- `&a` is the address where you want to store the variable (a was declared as `int a;`)
- To **write** to *stdout*:
 - `printf("The value is %d\n", a)`
 - `%d` means you want to write an integer
 - `\n` means you want to jump to a new line (aka "accapo")
- Other Options:

| Option | What you print |
|-----------------|--------------------------------------------------------|
| <code>%d</code> | int |
| <code>%c</code> | char |
| <code>%f</code> | float |
| <code>%s</code> | string (array of chars, <code>char*</code>) |
| <code>%x</code> | hexadecimal integer (for instance... 16 becomes 10) |
| <code>\n</code> | newline (no argument) |
| <code>\t</code> | tab (no argument) |

EXAMPLE

Read from Files, Write to Files

`stdio.h` lib (automatically included by compiler) provides you with the capability of interacting with FILES.

- Declares the FILE type
 - `FILE * file;`
- Lets you opening a file
 - `file = fopen("ciao.txt", "r" | "w" | "a");`
- To **read** from file(MUST be opened with the "r"-read option)
 - `fscanf("%d", &a);`
 - `%d =>` means you are reading an integer
 - `&a` is the address where you want to store the variable (a was declared as `int a;`)
- To **write** to a file (MUST be opened with the "w"-write or "a"-append option):

- `fprintf(file, "The value is %d\n", a)`
- `%d` means you want to write an integer
- `\n` means you want to jump to a new line (aka "accapo")
- Other Options:
 - Same as `stdin/stdout`

GCC Compiler Options

| Option | Description |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-c</code> | <p>Do NOT link.</p> <p>Useful in a multiple file project to compile single files separately (i.e. Counter .cpp). Linking step is usually performed later.</p> |
| <code>-g</code> | <p>Generate debug symbols. Without this, you cannot debug (KDBG, Eclipse etc).</p> <p>Drawback: file is bigger and code is slower since it is not optimized</p> |
| <code>-I<dir></code> | <p>At <i>compile</i> time, include a directory to look for included files (i.e. headers).</p> <p>By default, <code>-I.</code> (current dir) is added</p> |
| <code>-L<dir></code> | <p>At <i>linking</i> time, include a directory to look for object files/libraries: <code>.o</code>, <code>.a</code>.</p> <p>By default, <code>-L.</code> (current dir) is added</p> |
| <code>-l<name></code> | <p>At <i>linking</i> time, links <code>lib<name>.a</code> to the executable</p> |
| <code>-o</code> | <p>Output file name.</p> <p>If you compile a single file, use <code>SOURCENAME.o</code> as the object file name. If you are linking an entire executable (your full program), I strongly recommend <code>PROGNAME.x</code>, <code>.exe</code>. Default (ugly) name is <code>a.out</code></p> <p>Remember: "good" programming starts from the file name! A well written file name lets you to identify immediately the meaning of each file!!!</p> |
| <code>-O<n></code> | <p>Optimization level (<n> from 0 to 3)</p> |

Links to online resources

OOP and C++

<http://www.desy.de/gna/html/cc/Tutorial/tutorial.html>

<http://www.cplusplus.com> – Official, good and complete!!

<http://www.google.com> (***I'M NOT JOKING ...just use it, use it, use it!***)

Inheritance and polymorphism

<http://www.cplusplus.com/doc/tutorial/classes/>

<http://www.cplusplus.com/doc/tutorial/polymorphism/>

<http://www.ba.infn.it/~zito/jsem/lez3c.html>

I/O

<http://www.cplusplus.com/reference/clibrary/cstdio/printf/>